



Learn SQL Basics for Data Science

¿Qué son las bases de datos relacionales?

Las bases de datos relacionales son sistemas de gestión de datos que organizan la información en estructuras predefinidas llamadas **tablas**, las cuales se conectan entre sí mediante relaciones lógicas basadas en datos compartidos.

A continuación, se explica detalladamente su funcionamiento y componentes:

1. Estructura fundamental: Las Tablas

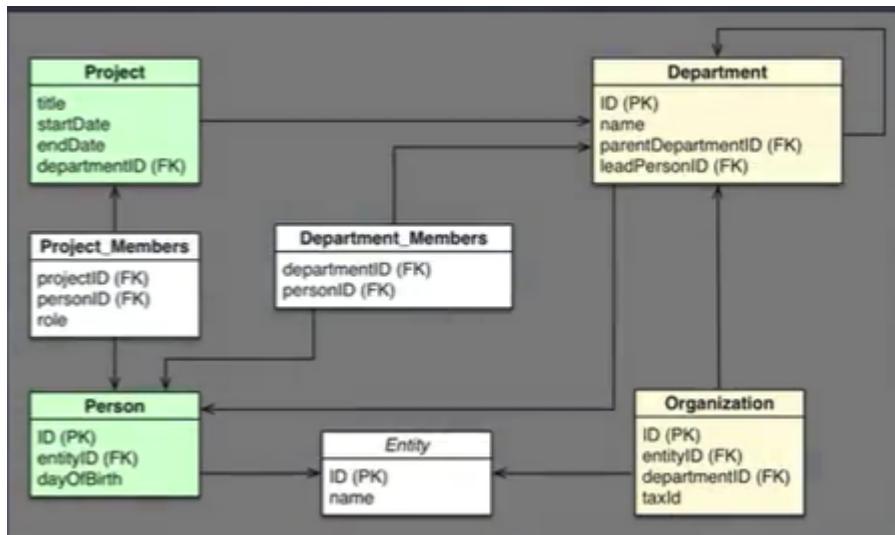
En el modelo relacional, los datos se almacenan en tablas compuestas por **filas** (también llamadas registros o instancias) y **columnas** (campos o atributos).

- **Columnas:** Cada columna tiene un nombre y un tipo de dato específico (como entero, cadena de texto o fecha).
- **Filas:** Cada fila representa un registro único dentro de la tabla.

2. Claves y Relaciones

El núcleo de una base de datos relacional es cómo se vinculan estas tablas para evitar la redundancia y mantener la integridad:

- **Clave Primaria (Primary Key):** Es un identificador único para cada registro en una tabla. Asegura que no haya filas duplicadas y que cada entidad pueda ser localizada de forma precisa.
- **Clave Foránea (Foreign Key):** Es una columna en una tabla que hace referencia a la clave primaria de otra tabla. Este es el mecanismo que crea el "vínculo" o relación entre los datos de diferentes tablas



Principales Statements

Realizar comentarios en SQL

Para una sola línea usaremos `--` y para más de una entre `/* */`

```
SELECT shoe_id
-- ,brand_id
,shoe_name
from shoes
```

```
SELECT shoe_id
/* ,brand_id
,shoe_name
*/
from shoes
```

CREATE TABLE

Nos permite crear nuestra tabla con sus instancias y atributos. Al crearla, hay que determinar el tipo de dato que vamos a insertar en cada columna, además de la PRIMARY KEY. Su forma es:

```
CREATE TABLE nombredetabla (nombre columnas tipo de dato)
```

```
CREATE TABLE Shoes
(
    Id          char(10)      PRIMARY KEY,
    Brand       char(10)      NOT NULL,
    Type        char(250)     NOT NULL,
    Color       char(250)     NOT NULL,
    Price       decimal(8,2)   NOT NULL,
    Desc        Varchar (750) NULL
);
```



Como vemos hay que especificar en cada línea si puede o no puede ser NULL. La PRIMARY KEY NUNCA puede ser NULL.

CREATE TEMPORARY TABLE

Las tablas temporales serán eliminadas cuando la sesión actual se cierre.

Es más rápido que crear una tabla real por lo que serán útiles para consultas usando subsets y joins.

Su forma es:

```
CREATE TEMPORARY TABLE nombre de tabla AS (consulta con los datos con lo que creamos la tabla)
```

```
CREATE TEMPORARY TABLE Sandals AS
(
    SELECT *
    FROM shoes
    WHERE shoe_type = 'sandals'
)
```

INSERT INTO

Nos permite insertar una instancia en una tabla. Su forma es:

```
INSERT INTO nombre de tabla VALUES (valores por columna)
```

```
INSERT INTO Shoes  
VALUES ('14535974'  
, 'Gucci'  
, 'Slippers'  
, 'Pink'  
, '695.00'  
, NULL  
);
```



Tendremos que incluir los valores en el orden en el que hemos creado las columnas.

Realizar consultas

SELECT FROM

Nos realizar una consulta a una base de datos. Su forma es:

```
SELECT nombre columna, nombre columna FROM nombre tabla
```

Si quisieramos obtener todo, sin distinguir entre columnas:

```
SELECT * FROM nombre tabla
```

```
SELECT column_name, column_name  
FROM table_name
```

Aspecto clave de las consultas

Cuando ejecutamos una consulta en SQL, el SELECT se ejecuta al final .Estoy hay que tenerlo en cuenta a la hora de utilizar funciones de agregacion, puesto que no podremos usar alias en la consulta ya que no se habrán ejecutado cuando llegue a la función de agregación:

```
SELECT a.nombre, COUNT(m.id) as total_medallas  
FROM atletas a
```

```
JOIN medallas m ON a.id = m.atleta_id  
GROUP BY a.nombre  
HAVING COUNT(m.id) > 1;
```

En el HAVING, no podría haber usado total_medallas porque la línea SELECT aun no se habría ejecutado cuando llegamos.



El ORDER BY y el LIMIT son las únicas funciones que se ejecutan después del SELECT, por lo que si que podremos usar alias para ordenar nuestra consulta.

WHERE

Nos permite filtrar la consulta atendiendo a una condición. Para ello:

```
SELECT * FROM nombre tabla WHERE nombre columna operator value
```

Los operators puede tener diversas formas, y por tanto expresar diferentes condiciones:

Operator	Description
=	Equal
<>	Not equal. Note: In some versions of SQL this operator may be written as !=
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
IS NULL	Is a null value

```
SELECT ProductName  
,UnitPrice  
,SupplierID  
FROM Products  
WHERE ProductName = 'Tofu';
```

Otros operadores que pueden ayudarnos a filtrar nuestras consultas son:

IN

Permite seleccionar instancias con la condición de que se encuentren en una lista de valores:

SELECT ProductID ,UnitPrice ,SupplierID From Products WHERE SupplierID IN (9, 10, 11);			
	ProductID	UnitPrice	SupplierID
1	22	21	9
2	23	9	9
3	24	4.5	10
4	25	14	11
5	26	31.23	11
6	27	43.9	11

Permite buscar dentro de otras consultas → IN (SELECT * FROM tabla)

OR

Establece dos o más posibles condiciones arrojando la consulta que cumple cualquiera de las dos. Si cumple la primera, arrojará esa consulta, por lo que habrá que tener cuidado con los paréntesis.

ProductID	UnitPrice	SupplierID	ProductName
1	14	23.25	6 Tofu


```
SELECT
    ProductName
    ,ProductID
    ,UnitPrice
    ,SupplierID
    ,ProductName
    From Products
    Where ProductName = 'Tofu' OR
    'Konbu';
```

AND

NOT

Permite unir más de una condición:

ProductID	UnitPrice	SupplierID
1	22	21
2	26	31.23
3	27	43.9

```
SELECT
    ProductID
    ,UnitPrice
    ,SupplierID
FROM Products
WHERE (SupplierID = 9 OR
SupplierID = 11)
AND UnitPrice > 15;
```

Niega una condición. Devolverá los valores que no la cumplan:

```
SELECT *
FROM Employees
WHERE NOT City='London' AND
NOT City='Seattle';
```

BETWEEN

Nos permite establecer una condición entre un rango de valores, string o fecha.

```
SELECT nombre, salario
FROM empleados
WHERE salario BETWEEN 30000 AND 50000;
```

El between funciona como un menor o igual, por lo que los números también estarán incluidos.

LIKE

Nos va a permitir filtrar las consultas atendiendo a la morfología de las palabras. Por tanto:

- Usaremos lo que se conoce como Wildcards para determinar el filtrado.
- Solo se podrá aplicar sobre columnas con tipo de datos String.
- No filtrará los NULL (puesto que no hay valor).
- **Tiene que ir siempre detrás de un WHERE.**

Wildcard	Action	Wildcard	Action
'%Pizza'	Grabs anything ending with the word pizza	'S%E'	Grabs anything that starts with "S" and ends with "E" (Like Sadie)
'Pizza%'	Grabs anything after the word pizza	't%@gmail.com'	Grabs gmail addresses that start with "t" (hoping to find Tom)
'%Pizza%'	Grabs anything before and after the word pizza		

Tienen la siguiente fórmula:

```
WHERE size LIKE '%pizza'
```

Output:

```
spizza  
mpizza
```



El símbolo % es equivalente a _

ORDER BY

Nos permite ordenar los datos según columnas específicas. Su fórmula:

```
SELECT something  
FROM database  
ORDER BY characteristic
```



Tiene que estar siempre al final de la consulta.

En este ejemplo estaríamos consultando los datos de la columna "something" de la base de datos "database" ordenados según la columna "characteristic". Si:

- DESC → Orden descendente.

- ASC → Orden ascendente.

Podemos ordenar por más de una columna, para que en caso de que haya empate en el primer criterio se desempate por el segundo. Por ejemplo, aquí ordenamos por el número de cursos pero si hay dos estudiantes con el mismo número de cursos se desempata por el orden alfabético del nombre:

```
SELECT e.nombre, COUNT(i.id_inscripcion) as total_cursos  
FROM estudiantes e  
LEFT JOIN inscripciones i  
ON e.id_estudiante = i.id_estudiante  
GROUP BY e.nombre  
ORDER BY total_cursos DESC, e.nombre ASC
```

Operaciones matemáticas

SQL nos permite realizar operaciones matemáticas en nuestras consultas. Por ejemplo, aquí estaríamos seleccionando dos columnas y creando una nueva resultante de la multiplicación entre ellas:

```
SELECT  
ProductID  
, UnitsOnOrder  
, UnitPrice  
, UnitsOnOrder * UnitPrice AS Total_Order_Cost  
FROM Products
```

Aggregate Functions

Nos permite ejecutar distintas funciones sobre nuestras consultas:

Function	Description
AVG ()	Averages a column of values
COUNT ()	Counts the number of values
MIN ()	Finds the minimum value
MAX ()	Finds the maximum value
SUM ()	Sums the column values

Por ejemplo, aquí estaríamos consultando la media de la columna "UnitPrice" y guardandola en la variable avg_price:

```
SELECT AVG(UnitPrice) AS avg_price
FROM products
```

COUNT ()

La función COUNT(), nos permite contar el número de valores (incluyendo los nulos o no):

COUNT (*) – Counts all the rows in a table containing values or NULL values

```
SELECT COUNT (*) AS
total_customers
FROM Customers;
```

Count (column) – Counts all the rows in a specific column ignoring NULL values

```
SELECT COUNT(CustomerID) AS
total_customers
FROM Customers
```

Como vemos, es normal poner un alias a la consulta puesto que sino nos dará count() como nombre de la columna.

DISTINCT ()

La función DISTINCT nos permite tener en cuenta solo aquellos valores que son distintos dentro de la columna, por lo que nos permitirá eliminar los duplicados:

```
SELECT COUNT(DISTINCT CustomerID)  
FROM Customers
```

LIMIT ()

La función LIMIT nos permite delimitar el número de resultados que queremos arrojar con nuestra consulta y la función OFFSET nos permite seleccionar a partir de qué resultado devolver los datos de nuestra consulta.

```
LIMIT num_limit OFFSET num_offset;
```

ALIAS

Nos permite dar un nombre temporal a una tabla o a una columna. Hacen las columnas más legibles. Solo existen durante la duración de la consulta.

```
SELECT vendor_name  
,product_name  
,product_price  
FROM Vendors, Products  
WHERE Vendors.vendor_id = Products.vendor_id;
```

```
SELECT vendor_name  
,product_name  
,product_price  
FROM Vendors AS v, Products AS p  
WHERE v.vendor_id = p.vendor_id;
```

Sin alias.

GROUP BY

Nos permite agrupar consultas según una columna específica. Por ejemplo, aquí estariamos mostrando el número de clientes que hay en cada región:

```
SELECT  
Region  
,COUNT(CustomerID) AS total_customers  
FROM Customers  
GROUP BY Region;
```

Para este statement hay que recalcar:

- Podemos agrupar por varias columnas.
- Cada columna en nuestro SELECT tendrá que estar presente en un GROUP BY.
- Los valores NULLS se agruparán juntos y la columna de agrupación que elegimos contiene NULLs.

Uno de los aspectos claves del GROUP BY es que obliga a incluir todos los atributos de la línea SELECT que no estén calculados (que no contengan una función de agregación).

HAVING

Nos permitirá poner condiciones a las agrupaciones hechas por el GROUP BY. Por ejemplo, aquí solo se mostrará el número para aquellos grupos en los que haya más de dos registros en cada grupo de CustomerID:

```
SELECT  
CustomerID  
,COUNT (*) AS orders  
FROM Orders  
GROUP BY CustomerID  
HAVING COUNT (*) >=2;
```

Estas son las principales diferencias con la cláusula WHERE:

	WHERE	HAVING
When It Filters	Values BEFORE Grouping	Values AFTER Grouping
Operates On Data From	Individual Rows	Aggregated Values from Groups of Rows
Example	SELECT username, followers FROM instagram_data WHERE followers > 1000;	SELECT country FROM instagram_data GROUP BY country HAVING AVG(followers) > 100;

Summary

Clause	Description	Required	
SELECT	Columns or expressions to be returned	Yes	Complete SELECT query SELECT DISTINCT column, AGG_FUNC(column_or_expression), ... FROM mytable JOIN another_table ON mytable.column = another_table.column WHERE constraint_expression GROUP BY column HAVING constraint_expression ORDER BY column ASC/DESC LIMIT count OFFSET COUNT;
FROM	Table from which to retrieve data	Only if selecting data from a table	
WHERE	Row-level filtering	No	
GROUP BY	Group specification	Only if calculating aggregates by group	
HAVING	Group-level filter	No	
ORDER BY	Output sort order	No	

CASE

Es la lógica if - then - else dentro de una consulta SQL. Es una de las herramientas más potentes para transformar datos en bruto en información con significado.

Sintaxis

Un bloque CASE siempre sigue una estructura lógica cerrada: busca la primera coincidencia, devuelve el resultado y se detiene.

CASE

```
WHEN condicion_1 THEN 'Resultado A'  
WHEN condicion_2 THEN 'Resultado B'  
ELSE 'Resultado por defecto'  
END AS nombre_columna
```

- **WHEN**: Define la condición a evaluar.
- **THEN**: Es lo que se muestra si la condición es verdadera.
- **ELSE (opcional)**: Lo que se muestra si ninguna condición anterior se cumplió. Si no lo pones y nada coincide, devolverá **NULL**.
- **END**: Indica que la lógica ha terminado. **Es obligatorio**.

Usos principales en el análisis de datos

A. Categorización (Crear etiquetas)

Es el uso más común. Sirve para agrupar valores numéricos en grupos fáciles de entender.

- *Ejemplo:* Si la producción es > 500M es "Líder", si no, es "Soporte".

B. Limpieza de datos (Sustitución)

Ya lo mencionamos antes: sirve para cambiar valores inconsistentes.

- *Ejemplo:* `CASE WHEN Value = ' (D)' THEN '0' ELSE Value END`.

C. Agregación Condicional +

Este es el nivel avanzado. Nos permite "pivotar" datos o contar cosas específicas en columnas separadas sin usar filtros que eliminan filas.

En lugar de contar todo, le decimos a SQL: "*Suma 1 solo si se cumple esta condición, si no, suma 0*".

Ejemplo práctico

Clasificación de registros de una tabla según la producción:

```

SELECT
State,
Year,
Value,
CASE
WHEN Value > 500000000 THEN '🔴 Producción Masiva'
WHEN Value > 100000000 THEN '🟡 Producción Media'
ELSE '🟢 Producción Baja'
END AS categoria_volumen -- solemos poner otro nombre para distinguirlo de
la columna original
FROM yogurt_production;
```

Subconsultas

Son consultas dentro de otras consultas. Permitidas y útiles en bases de datos relacionales donde la información se guarda en diferentes tablas relacionadas entre ellas. Permite unir data de múltiples fuentes y ayudan añadiendo otros filtros de datos.

```

SELECT
CustomerID
,CompanyName
,Region
FROM Customers
WHERE customerID in (SELECT
customerID
FROM Orders
WHERE Freight > 100 );

```

Dentro de sus principales características destacamos:

- No hay límite en cuanto a subconsultas que se pueden realizar.
- Si las nesteamos muy profundamente el rendimiento de la consulta será bajo (en cuanto a rapidez).
- El SELECT de la subconsulta solo puede hacer referencia a una única columna o a un único valor.

Un ejemplo → Conseguir los nombres y los detalles de contacto de todos los clientes que han ordenado pasta de dientes:

```

SELECT Customer_name, Customer_contact
FROM Customers
WHERE cust_id IN
    (SELECT customer_id
     FROM Orders
     WHERE order_number IN
          (SELECT order_number
           FROM OrderItems
           WHERE prod_name = 'Toothbrush'));

```

Otro ejemplo → Obtener todos los estados con una producción superior a la media de ese año:

```
SELECT
s.State,
y.value as produccion_2022
FROM state_lookup s
JOIN yogurt_production y ON s.State_ANSI = y.State_ANSI
WHERE y.year = 2022
AND y.value > (
SELECT AVG(value)
FROM yogurt_production
WHERE year = 2022
);
```

En esta consulta, primeo se ejecuta lo que está entre paréntesis (la subconsulta) la cual arroja un número. Luego, la consulta recorre la tabla principal buscando que estados en 2022 superaron ese número.



La subconsultas cuando se encuentras siguiendo a la clausula FROM exigen un alias obligatorio. Si están detrás de otra (como WHERE o SELECT) no la exigen pero es recomendable.

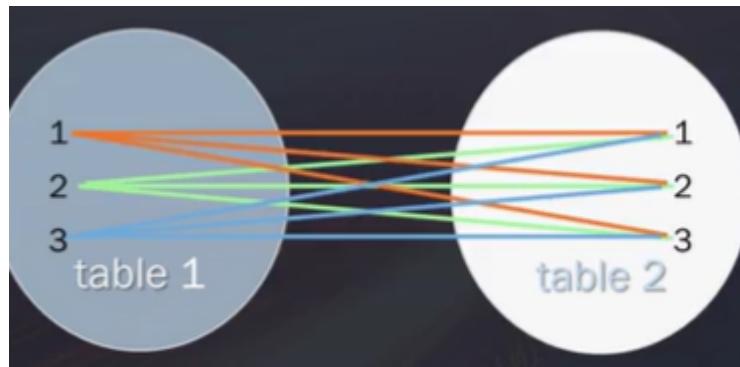
JOINS

Permiten asociar instancias de distintas tablas. Permite consultar datos de diferentes tablas en una sola consulta. No son físicos, es decir, solo persisten en la duración en la que se hace la consulta.

Habrá que especificar el TIPO de join y **la COLUMNA de union entre las dos tablas.**

Cartesian (Cross) Joins

Aquella relación en la que cada fila de la primera tabla se relaciona con todas las filas de la otra tabla.



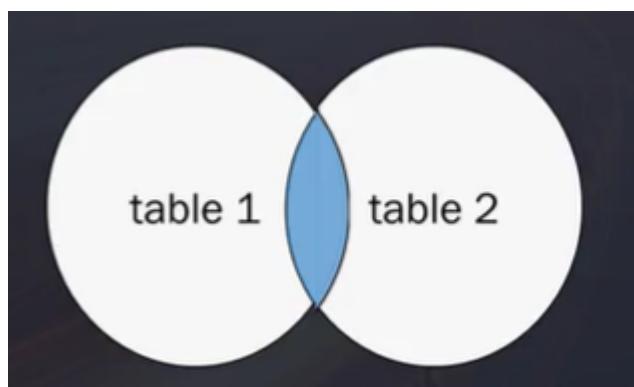
Un ejemplo:

```
SELECT product_name
,unit_price
,company_name
FROM suppliers CROSS JOIN products;
```

El output corresponderá con el número de joins en la primera tabla multiplicado por el número de filas de la segunda tabla.

Inner Joins

Selecciona instancias que tengan valores comunes en las dos tablas.



Un ejemplo sería el siguiente:

```

SELECT column, another_table_column, ...
FROM mytable
INNER JOIN another_table
  ON mytable.id = another_table.id
WHERE condition(s)
ORDER BY column, ... ASC/DESC
LIMIT num_limit OFFSET num_offset;

```

```

SELECT suppliers.CompanyName
,ProductName
,UnitPrice
FROM Suppliers INNER JOIN Products
ON Suppliers.supplierid =
Products.supplierid

```

En este ejemplo, vamos a obtener “companyName” de la tabla suppliers y productName y UnitPrice de la tabla productos donde el id del suplier coincide con la FK de supplier id en la tabla productos.

Self Join

Une la tabla original con ella misma. Un ejemplo sería encontrar todos los trabajadores con sus respectivos managers:

```

SELECT
    e1.FirstName || ' ' || e1.LastName
AS EmployeeName,
    e2.FirstName || ' ' || e2.LastName
AS ManagerName
FROM Employee e1
LEFT JOIN Employee e2 ON e1.ReportsTo
= e2.EmployeeId
ORDER BY EmployeeName;

```

Left Join

Devuelve todas las instancias de la tabla de la izquierda y y los comunes con la segunda tabla. El resultado es NULL para el lado derecho si no hay match.



Un caso muy característico de este tipo de JOIN es obtener los datos de una tabla que tiene valores vacíos en la otra tabla (IS NULL). Por ejemplo, si queremos saber que estudiantes no estan inscritos en ningun curso:

```
SELECT e.nombre  
FROM estudiantes e  
LEFT JOIN inscripciones i  
ON e.id_estudiante = i.id_estudiante  
WHERE i.nombre_curso IS NULL;
```



El método más robusto es usar la PK de la tabla a la que nos unimos
→ WHERE i.id_inscripcion IS NULL

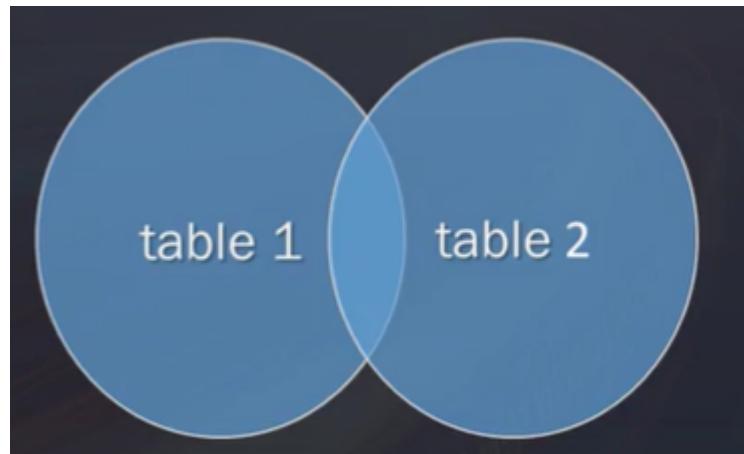
Right Join

Devuelve todos los resultados de la tabla de la derecha más aquellos que coincidan con la tabla izquierda. Igual que antes, devuelve NULL de la tabla izquierda si no hay match.



Full Outer Join

Devuelve todos los resultados de ambas tablas.



Consideraciones

Los joins son operaciones costosas de procesas así que:

- Selecciona campos específicos en vez de usar SELECT *.
- Usar LIMIT para reducir las consultas.
- Filtrar y agregar antes de hacer el JOIN.
- Evitar multiples JOINS en una misma consulta.

UNION

Se usa para combinar los resultados de dos o mas SELECT statements.

Cada SELECT statament dentro de la UNION tiene que tener el mismo número de columnas, además de tipos de datos similares y estar en el mismo orden.

```

SELECT City, Country FROM
Customers
WHERE Country='Germany'
UNION
SELECT City, Country FROM
Suppliers
WHERE Country='Germany'
ORDER BY City;
    
```

Resumen



Working with String Variables

Las funciones de modificación de data String nos permite consultar los datos en el formato que nosotros queremos, pudiendolo adecuar a las necesidades del clientes.

Soportan joins.

Algunas de las principales son:

- Concatenación: nos permite unir las instancias de dos columnas en una misma:

```

SELECT
  CustomerId,
  FirstName || ' ' || LastName AS FullName,
  Address,
  UPPER(City || ' ' || Country) AS CityCountry
FROM customers;
  
```

CityCountry
SÃO JOSÉ DOS CAMPOS BRAZIL
STUTTGART GERMANY
MONTRÉAL CANADA
OSLO NORWAY
PRAGUE CZECH REPUBLIC
VIENNE AUSTRIA
BRUSSELS BELGIUM
COPENHAGEN DENMARK

- Substring: devuelven un número específico de caracteres de una particular posición de un string dato.

<code>SUBSTR(string name, string position, number of characters to be returned);</code>	<code>First_name</code>	<code>substr(first_name,2,3)</code>
	Alexander	lex
	Bruce	ruc
	David	avi
<code>SELECT first_name, SUBSTR(first_name,2,3)</code>	Valli	all
<code>FROM employees</code>	Diana	ian
<code>WHERE department_id=60;</code>		

- Upper and Lower: pone mayúsculas o las quita

```
SELECT UPPER(column_name) FROM table_name;
```



```
SELECT LOWER(column_name) FROM table_name;
```

Working with DateTime variables

Existen varios formatos para expresar medidas de tiempo, que habrá que reseñar si queremos que nuestras consultas nos devuelvan resultados:

DATE
Format YYYY-MM-DD
DATETIME
Format: YYYY-MM-DD HH:MI:SS
TIMESTAMP
Format: YYYY-MM-DD HH:MI:SS

Estas son las 5 funciones que soporta SQL en cuanto a datos temporales junto con sus modificadores:

```

DATE(timestamp, modifier, modifier, ...)
TIME(timestamp, modifier, modifier, ...)
DATETIME(timestamp, modifier, modifier, ...)
JULIANDAY(timestamp, modifier, modifier, ...)
STRFTIME(format, timestamp, modifier, modifier, ...)

```

NNN days	start of year
NNN hours	start of day
NNN minutes	weekday N
NNN.NNNN seconds	unixepoch
NNN months	localtime
NNN years	utc
start of month	

Estas son algunos ejemplos de consultas que podemos hacer con datos temporales:

- Separar los componentes de una fecha

Birthdate	Year	Month	Day
1 1962-02-18 00:00:00	1962	02	18
2 1958-12-08 00:00:00	1958	12	08
3 1973-08-29 00:00:00	1973	08	29
4 1947-09-19 00:00:00	1947	09	19
5 1965-03-03 00:00:00	1965	03	03
6 1973-07-01 00:00:00	1973	07	01
7 1970-05-29 00:00:00	1970	05	29
8 1968-01-09 00:00:00	1968	01	09

```

SELECT Birthdate
,STRFTIME('%Y', Birthdate) AS Year
,STRFTIME('%m', Birthdate) AS Month
,STRFTIME('%d', Birthdate) AS Day
FROM employees

```

- Seleccionar la fecha y hora actual:

```
SELECT STRFTIME('%Y %m %d', 'now')
```

- Calcular la edad a partir de una fecha:

```

SELECT Birthdate
,STRFTIME('%Y', Birthdate) AS Year
,STRFTIME('%m', Birthdate) AS Month
,STRFTIME('%d', Birthdate) AS Day
,STRFTIME('now') - Birthdate AS Age
FROM employees

```

Extract

Esta función nos permite extraer una parte de la fecha para usarla a modo de filtrado. Por ejemplo, si queremos filtrar las instancias que ocurrieron antes de un mes en concreto:

```
SELECT *
FROM partidos
WHERE EXTRACT(MONTH FROM fecha_evento) = 12;
```

Si queremos mes y día:

```
SELECT *
FROM partidos
WHERE EXTRACT(MONTH FROM fecha_evento) = 12 -- Que sea Dic
iembre
      AND EXTRACT(DAY FROM fecha_evento) = 25; -- Que sea el
día 25
```

Case Statement

Permiten usar bloques IF - THEN - ELSE como en otros lenguajes de programación. Se pueden usar en SELECT, INSERT, UPDATE y DELETE.

```
CASE input_expression
    WHEN when_expression THEN result_expression [ ...n ]
    [ ELSE else_result_expression ]
END
```

Un ejemplo práctico sería el siguiente:

```

SELECT
    employeeid
    ,firstname
    ,lastname
    ,city
    ,CASE City
        WHEN 'Calgary' THEN 'Calgary'
    ELSE 'Other'
    END calgary
FROM Employees
ORDER BY LastName, FirstName;

```

	employeeid	firstname	lastname	city	calgary
1	1	Andrew	Adams	Edmonton	Other
2	8	Laura	Callahan	Lethbridge	Other
3	2	Nancy	Edwards	Calgary	Calgary
4	5	Steve	Johnson	Calgary	Calgary
5	7	Robert	King	Lethbridge	Other
6	6	Michael	Mitchell	Calgary	Calgary
7	4	Margaret	Park	Calgary	Calgary
8	3	Jane	Peacock	Calgary	Calgary

Views

Nos permiten crear “vistas” con columnas de tablas que nosotros queramos permitiendo tratarlas como si fueran tablas físicas donde poder realizar consultas.

Aquí estaríamos creando una vista:

Creating a View

```

CREATE VIEW my_view
AS
SELECT
    r.regiondescription
    ,t.territorydescription
    ,e.Lastname
    ,e.Firstname
    ,e.Hiredate
    ,e.Reportsto
FROM Region r
INNER JOIN Territories t on r.regionid = t.regionid
INNER JOIN EmployeeTerritories et on t.TerritoryID = et.TerritoryID
INNER JOIN Employees e on et.employeeid = e.EmployeeID

```

regiondescription	territorydescription	Lastname	Firstname	Hiredate	Reportsto
1 Eastern	Wilton	Davolio	Janet	5/1/1992 12:00:00 AM	2
2 Eastern	Neword	Davolio	Janet	5/1/1992 12:00:00 AM	2
3 Eastern	Westboro	Fuller	Andrew	8/14/1992 12:00:00 AM	NULL
4 Eastern	Bedford	Fuller	Andrew	8/14/1992 12:00:00 AM	NULL
5 Eastern	Georgetown	Fuller	Andrew	8/14/1992 12:00:00 AM	NULL
6 Eastern	Boston	Fuller	Andrew	8/14/1992 12:00:00 AM	NULL
7 Eastern	Cambridge	Fuller	Andrew	8/14/1992 12:00:00 AM	NULL
8 Eastern	Braintree	Fuller	Andrew	8/14/1992 12:00:00 AM	NULL
9 Eastern	Louisville	Fuller	Andrew	8/14/1992 12:00:00 AM	NULL
10 Southern	Atlanta	Leverling	Janet	4/1/1992 12:00:00 AM	2
11 Southern	Savannah	Leverling	Janet	4/1/1992 12:00:00 AM	2
12 Southern	Orlando	Leverling	Janet	4/1/1992 12:00:00 AM	2
13 Southern	Tampa	Leverling	Janet	4/1/1992 12:00:00 AM	2
14 Eastern	Rockville	Peacock	Margaret	5/1/1993 12:00:00 AM	2
15 Eastern	Greensboro	Peacock	Margaret	5/1/1993 12:00:00 AM	2
16 Eastern	Cay	Peacock	Margaret	5/1/1993 12:00:00 AM	2
17 Eastern	Providence	Buchanan	Steven	10/17/1993 12:00:00 AM	2
18 Eastern	Manistown	Buchanan	Steven	10/17/1993 12:00:00 AM	2
19 Eastern	Edison	Buchanan	Steven	10/17/1993 12:00:00 AM	2
20 Eastern	New York	Buchanan	Steven	10/17/1993 12:00:00 AM	2
21 Eastern	New York	Buchanan	Steven	10/17/1993 12:00:00 AM	2
22 Eastern	Melville	Buchanan	Steven	10/17/1993 12:00:00 AM	2
23 Eastern	Fairport	Buchanan	Steven	10/17/1993 12:00:00 AM	2
24 Western	Phoenix	Sayama	Michael	10/17/1993 12:00:00 AM	5
25 Western	Glendale	Sayama	Michael	10/17/1993 12:00:00 AM	5

De la que luego podríamos realizar consultas:

	count(territorydescription)	Lastname	Firstname
1	7	Buchanan	Steven
2	4	Callahan	Laura
3	2	Davolio	Nancy
4	7	Dodsworth	Anne
5	7	Fuller	Andrew
6	10	King	Robert
7	4	Leverling	Janet
8	3	Peacock	Margaret
9	5	Suyama	Michael

```
SELECT SUM(y.Value)
FROM yogurt_production y
WHERE y.Year = 2022 AND y.State_ANSI IN (
```

```
    SELECT DISTINCT c.State_ANSI FROM cheese_production c WHERE
    c.Year = 2022
```

);

3

```
SELECT AVG(Value) FROM honey_production WHERE Year = 2022;
```

1

Data cleaning

Entre las principales funciones de limpieza de datos destacan:

Función	¿Qué hace?	Ejemplo 
<code>REPLACE(col, 'a', 'b')</code>	Cambia un carácter por otro.	Quitar las comas <code>,</code> de un número.
<code>TRIM(col)</code>	Borra espacios vacíos al inicio o final.	Limpiar <code>" Nueva York "</code> → <code>"Nueva York"</code> .
<code>COALESCE(col, 0)</code>	Reemplaza valores nulos (<code>NULL</code>) por uno fijo.	Si no hay dato, poner un <code>0</code> .
<code>CAST(col AS tipo)</code>	Cambia el tipo de dato (ej. de texto a número).	<code>"123"</code> (texto) → <code>123</code> (entero).
<code>UPPER() / LOWER()</code>	Pone todo en mayúsculas o minúsculas.	Estandarizar <code>"California"</code> y <code>"california"</code> .

Cuando nos encontramos con una columna “sucia”, este es el orden lógico a seguir en las funciones:

Paso	Acción	Herramienta
1. Identificar	Detectar textos en columnas numéricicas (como 'N/A' o '(D)').	<code>REPLACE</code> o <code>CASE</code>
2. Estandarizar	Quitar símbolos de moneda, comas o puntos.	<code>REPLACE</code>
3. Pulir	Eliminar espacios accidentales al inicio o final.	<code>TRIM</code>
4. Convertir	Cambiar el tipo de dato de texto a número.	<code>CAST</code> o <code>CONVERT</code>

Window Functions

PARTITION BY

Permite dividir la consulta en mini grupos atendiendo a un atributo o columna. Por ejemplo, si usamos PARTITION BY Year, SQL crea un grupo separado para cada año, y cualquier calculo que hagamos se reiniciará cada vez que empezemos un grupo. Implica un OVER previo.

RANK () OVER

Esta función asigna un número de posición a cada fila dentro de su grupo.

- Para decidir quién es el #1, necesitamos un ORDER BY.
- Si hay un empate (dos filas tienen el mismo valor), RANK () les otorga el mismo número, pero saltará al siguiente (1, 2, 2, 4). Para evitar esto, hay otra función, DENSE_RANK () que no deja esos huecos (1, 2, 2, 3) o ROW_NUMBER() que no permite empates, simplemente asigna un número único según el orden que los encuentre.

Un ejemplo de estas dos funciones en acción:

Year	State	Value	RANK() OVER (PARTITION BY Year ORDER BY Value DESC)
2022	NY	793M	1 (El más alto de 2022)
2022	CA	377M	2
2021	NY	774M	1 (¡El ranking se reinicia porque cambió el año!)
2021	CA	374M	2

Esta sería la principal diferencia entre las tres funciones:

ROW_NUMBER | RANK | DENSE_RANK

Popularity	Popularity_R	Popularity_DR
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	5	5
7	7	6
8	8	7
9	9	8
10	10	9

Un ejemplo claro de uso de una función de ventana sería el siguiente:

```
SELECT Gender, Name, Total,  
       ROW_NUMBER() OVER(ORDER BY Total DESC) AS Popularity  
FROM baby_names;
```

Boy	Noah	120	1
Boy	Ethan	115	2
Girl	Emma	106	3
Boy	Jacob	101	4
Girl	Isabella	100	5
Girl	Olivia	100	6
Girl	Ava	95	7
Girl	Sophia	88	8
Boy	Liam	84	9
Boy	Logan	73	10

Aquí tendríamos una función de ventana sin partición, que nos daría el número de fila para la ventana (todo nuestro data set).

Si ahora queremos saber el orden de popularidad dividido en chicos y chicas.

```
SELECT Gender, Name, Total,  
       ROW_NUMBER() OVER(PARTITION BY Gender ORDER BY Total DESC) AS Popularity  
FROM baby_names;
```

Gender	Name	Total	Popularity
Boy	Noah	120	1
Boy	Ethan	115	2
Boy	Jacob	101	3
Boy	Liam	84	4
Boy	Logan	73	5
Girl	Emma	106	1
Girl	Isabella	100	2
Girl	Olivia	100	3
Girl	Ava	95	4
Girl	Sophia	88	5

LAG () y LEAD ()

La función LAG () nos permite traer el valor de una fila anterior a una fila actual sin necesidad por ejemplo de hacer un SELF JOIN.

La estructura básica es `LAG(columna, desplazamiento, valor_por_defecto) OVER (...)`:

- **Columna:** El dato que quieras recuperar del pasado (ej. la producción de yogur).
- **Desplazamiento (opcional):** Cuántas filas hacia atrás quieras mirar. Por defecto es `1` (la fila inmediatamente anterior).
- **Valor por defecto (opcional):** Qué mostrar si no hay una fila anterior (por ejemplo, en la primera fila de la tabla). Suele usarse `0` o `NULL`.
- **OVER (ORDER BY ...):** Es **obligatorio**. SQL necesita saber cuál es el orden cronológico o lógico para decidir qué fila es la "anterior". 

Un ejemplo, usando esta función sería la siguiente:

```
SELECT
Year,
value AS produccion_actual,
LAG(value) OVER (ORDER BY Year) AS produccion_anterior,
(value - LAG(value) OVER (ORDER BY Year)) AS diferencia_neta
FROM yogurt_production
WHERE State_ANSI = 36;
```

La función LEAD () haría lo contrario, nos permitiría traer el valor de una fila posterior a la fila actual.

CTEs

Las CTEs (Common Table Expressions) o Expresiones de Tabla Común son como "tablas virtuales" temporales que definimos justo antes de ejecutar nuestra consulta principal. Se crean utilizando la palabra clave WITH.

Sus principales ventajas:

- **Legibilidad:** Permiten dividir consultas gigantes en trozos lógicos y fáciles de leer. En lugar de tener subconsultas dentro de subconsultas, defines cada paso por su nombre.
- **Modularidad:** Puedes definir varias CTEs seguidas separadas por comas y luego usarlas todas en tu `SELECT` final. Incluso una CTE puede hacer referencia a otra definida anteriormente.
- **Ámbito Temporal:** Solo existen durante la ejecución de esa consulta específica. No ocupan espacio permanente en la base de datos como lo haría una tabla real

Sintaxis

La estructura de una CTE es bastante lógica y se lee casi como una oración. Aquí tienes los componentes principales que debemos usar:

Parte	Función
WITH	La palabra clave que le indica a la base de datos: "Atención, voy a preparar una tabla temporal!" 
Nombre_de_la_CTE	El nombre que tú elijas para identificar estos datos (por ejemplo: <code>produccion_nacional</code>) 
AS (...)	Dentro de los paréntesis escribes el <code>SELECT</code> que "cocina" los datos iniciales 
Consulta Principal	El <code>SELECT</code> final que utiliza la CTE que acabas de definir como si fuera una tabla física 

En código, la plantilla básica se ve así:

```
WITH nombre_de_tu_cte AS (
  SELECT columna1, SUM(columna2) as total
  FROM tabla_original
```

```

GROUP BY columna1
)
SELECT * FROM nombre_de_tu_cte; -- Aquí usas la tabla que creaste arriba

```



Hay que tener en cuenta que luego haremos consultas a partir de esa "tabla virtual" que hemos creado por lo que será necesario seleccionar todas las columnas de las que luego queramos hacer uso.

CTEs vs Subconsultas

Característica	Subconsulta (Subquery)	CTE (WITH)
Legibilidad	Puede volverse confusa si hay mucha anidación ("código espagueti"). 🍝	Estructura limpia y secuencial, de arriba hacia abajo.
Reutilización	Solo se puede usar una vez dentro de la consulta donde se define. 🤪	Se puede llamar varias veces en la misma consulta principal. ↗
Mantenimiento	Difícil de depurar porque el error suele estar "escondido" en el centro. 🔎	Fácil de probar cada bloque por separado antes de unirlo todo. ✓
Recursividad	No permite hacer cálculos recursivos. ❌	Permite consultas recursivas (útil para organigramas). 🌱
Rendimiento	Los motores modernos suelen optimizarlas igual de bien. ⚡	Similar, aunque en algunos sistemas se pueden "materializar" (guardar en memoria). 💾

¿Cuándo usar cada una?

- **Usa Subconsultas** para cosas muy rápidas y simples, como un filtro pequeño en el `WHERE` (ej. `WHERE precio > (SELECT AVG(precio) FROM...)`).
- **Usa CTEs** siempre que tu lógica tenga varios pasos o cuando necesites calcular un valor (como un total anual) para luego usarlo en una operación más compleja. Es la mejor práctica para analistas de datos profesionales.